Compositional Learning for Interleaving Parallel Automata

Faezeh Labbaf[1][0000−0002−8812−6702], Jan Friso Groote[2][0000−0003−2196−6587], Hossein Hojjat[1,3][0000−0002−4743−8750], and Mohammad Reza Mousavi[4][0000−0002−4869−6794]

[1] Tehran Institute for Advanced Studies (TeIAS), Khatam University, Iran
f.labaf@khatam.ac.ir
[2] Eindhoven University of Technology, The Netherlands
j.f.Groote@tue.nl
[3] University of Tehran, Iran
hojjat@ut.ac.ir
[4] King's College London, UK
mohammad.mousavi@kcl.ac.uk

Abstract. Active automata learning has been a successful technique to learn the behaviour of state-based systems by interacting with them through queries. In this paper, we develop a compositional algorithm for active automata learning in which systems comprising interleaving parallel components are learned compositionally. Our algorithm automatically learns the structure of systems while learning the behaviour of the components. We prove that our approach is sound and that it learns a maximal set of interleaving parallel components. We empirically evaluate the effectiveness of our approach and show that our approach requires significantly fewer numbers of input symbols and resets while learning systems. Our empirical evaluation is based on a large number of subject systems obtained from a case study in the automotive domain.

1 Introduction

Active automata learning has been successfully used to learn models of complex industrial systems such as communication- and security protocols [11], biometric passports [2], smart cards [1], large-scale printing machines [33], and lithography machines for integrated circuits [32,15]; we refer to the recent survey by Howar and Steffen on the practical applications of active automata learning [16]. Throughout these applications of automata learning, scalability issues have been pointed out [32,15]. It has also been suggested that compositional learning, i.e., learning a system through learning its components, is a promising approach to tame the complexity of learning [10,12].

Some early attempts have been recently made in learning structured models of systems [27,10] (we refer to the Related Work for an in-depth analysis). For example, the approach proposed by al-Duhaiby and Groote [10] decomposes the learning process into learning its parallel components; however, it relies on a deep knowledge of the system under learning, and the intricate interaction of the various actions being learned. In this paper, we propose an approach based on Dana Angluin's celebrated L^* algorithm [6], to learn the components of a system featuring an interleaving parallel composition. Our approach, called CL^*, does not assume any pre-knowledge of the structure and the alphabet of these components; instead, we learn this information automatically and on-the-fly, while providing a rigorous guarantee of the learned information. This is particularly relevant in the context of legacy and black-box systems where architectural discovery is challenging [8,22].

The gist of our approach is to learn the System Under Learning (SUL) in separate components with disjoint alphabets. We start with a partition comprising only singleton sets. The interleaving

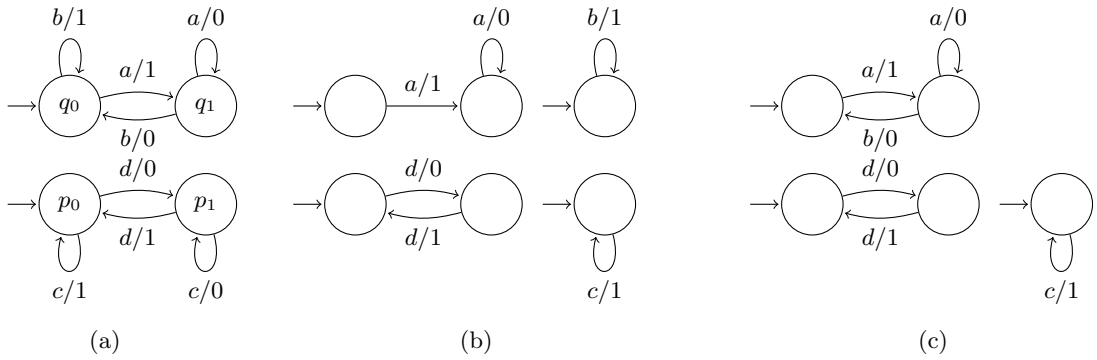

Fig. 1: (a) Initial system with two concurrent FSMs (b) Partition the input alphabet to 4 elements and learn each component individually (c) Use the counter-example ab to merge two components

parallel composition of the components gives us the total behavior of the system. We pass the result to the teacher, and by exploiting the counter-examples returned, we iteratively merge the alphabet of the individual components.

Example. Figure 1(a) shows an example of two parallel Finite State Machines (FSMs) over the input alphabet $\{a, b, c, d\}$ and output alphabet $\{0, 1\}$. We start by partitioning the alphabet into disjoint singleton sets of elements. The parallel composition of the 4 learned FSMs of Figure 1(b) does not comply with the original system, and the teacher may return the counter-example ab. The string ab generates the output sequence 10 in (a) but the output sequence in (b) is 11. The counter-example suggests to merge the sets $\{a\}$ and $\{b\}$ and restart the learning process which leads to the FSMs in Figure 1(c). One further merging step results in learning the original system. We provide a theoretical proof of correctness of this compositional construction, meaning that it is guaranteed to construct a correct system.

To study the effectiveness of our approach in practice, we designed an empirical experiment to investigate the following two research questions:

RQ1 Does CL* require fewer resets, compared to L*?
RQ2 Does CL* require fewer input symbols, compared to L*?

Our research questions are motivated by the following facts: 1) Resets are a major contributing factor in learning practical systems as they are immensely time- and resource consuming [31]. Hence, reducing the number of resets can have a significant impact in the learning process. 2) The total number of symbols used in interacting with the system under learning provides us with a total measure of cost for the learning process and hence, reducing the total cost is a fair indicator of improved efficiency [36,9].

To answer these questions, we use a benchmark based an industrial automotive system. We design a number of experiments on learning various combinations of components in this system, gather empirical data, and analyse them through statistical hypothesis testing. Our results indicate that our compositional approach significantly improves the efficiency of learning compared to the monolithic L* learning algorithm. The implementation of the algorithm, experiments, and their results can be found on-line in our lab package [23] (`https://github.com/faezeh-lbf/CL-Star`).

The remainder of this paper is organised as follows. In Section 2, we review the related work and position our research with respect to the state of the art. In Section 3, we present the preliminary definitions that are used throughout the rest of the paper. In Section 4, we present our algorithm and its proof of correctness and termination. We evaluate our algorithm on a benchmark from the automotive domain in Section 5. We conclude the paper and present the directions of our ongoing and future research in Section 6.

2 Related Work

Active automata learning is a technique used to find the underlying model of a black box system by posing queries and building a hypothesis in an iterative manner. There is substantial early work in this domain, e.g., under the name system identification or grammar inference; we refer to the accessible introduction by Vaandrager [36] for more information. A seminal work in this domain is the L^* algorithm by Dana Angluin [6], which comes with theoretical complexity bounds for the learning process using a representation called the "Minimally Adequate Teacher" (MAT).

MAT hypothesises a teacher that is capable of responding to membership queries (MQs) and equivalence queries (EQs); the former checks the outcome of a sequence of inputs (e.g., with their respective outputs, or with their membership in the language of the automaton) and the latter checks whether a hypothesised automaton is equivalent to the system under learning. Our work replaces a single MAT with multiple MATs that can potentially run in parallel and learn different components of the black-box system automatically.

Learning structured systems and in particular, compositional learning of parallel systems has been studied recently in the literature. Moerman [27] proposes an algorithm to learn parallel interleaving Moore machines. Our algorithm differs from Moerman's algorithm in that in the parallel composition of Moore machines, the output of each individual component is explicitly specified, because the output of the system is specified as a tuple of the outputs of its components. In other words, the underlying structure is immediately exposed by considering the type of outputs produced by the system under learning. However, in our approach, we need to identify the components and assign outputs to them on-the-fly since the decomposition is not explicit in parallel composition. Al-Duhaiby and Groote [10] learn parallel labelled transitions systems with the possibility of synchronisation among them. In order to develop their algorithm, they assume a priori knowledge of mutual dependencies among actions in terms of a confluence relation. This type of information is difficult to obtain and the domain knowledge in this regard may be error prone. Particularly for legacy and large black-box systems (e.g., binary code), architectural discovery has proven challenging [8,22]. We address this challenge and go beyond the existing approaches by learning about confluence of actions on-the-fly through observing the minimal counter-examples generated by the MAT(s).

Frohme and Steffen [12] introduce a compositional learning approach for Systems of Procedural Automata [13]; these are collections of DFAs that may "call" each, akin to the way non-terminals may be used in defining other non-terminals in a grammar. Their approach is essentially different from ours in that the calls across automata are assumed to be observable and hence the general structure is assumed to be known; in our approach, we learn the structure by observing implicit dependencies among the learned automata through analysing counter-examples. Also their approach is aimed at a richer and more expressive type of systems, namely pushdown systems, which justifies the requirement for additional information.

L^* has been improved significantly in the past few years; the major improvements upon L^* can be broadly categorised into three categories: 1) improving the data structures used to store and retrieve the learned information [21,31,19,37]; 2) improving the way counter-examples are processed in refining the hypothesis [31,28,3,17]; 3) learning more expressive models, such as register- [18,14] and timed automata [34,5]. This third category of improvements is orthogonal to our contribution and extension of our approach can be considered in those contexts as well.

Two notable recent improvements, in the first two categories, are $L^\#$ [37] and L^λ [17], respectively. $L^\#$ uses the notion of *apartness* to organise and maintain a tree-shaped data-structure about the learned automaton. L^λ uses a search-based method to incorporate the information about the counter-example into the learned hypothesis. The improvements brought about by L^λ can be readily incorporated into our approach, particularly since our approach relies on finding minimal counter-examples. Integrating our approach into $L^\#$ requires a more careful consideration of maintaining and composing tree-shaped data structures when detecting dependencies. We expect that both of these combinations will further improve the efficiency of our proposed method.

3 Preliminaries

In this section, we review the basic notions used throughout the remainder of the paper. We start by formalising the notion of a finite state machine, which is the underlying model of the system under learning and move on to parallel composition and decomposition (called projection) as well as the concept of (in)dependent actions, which are essential in identifying the parallel components. Finally, we conclude this section by recalling the basic concepts of active automata learning and the L^* algorithm.

3.1 Finite State Machines (FSMs)

Finite state machines (also called Mealy machines), defined below, are straightforward generalisations of finite automata in which the transitions produce outputs (rather than only indicating acceptance or non-acceptance):

Definition 1. *(Finite State Machine) A Finite State Machine (FSM) M is a sixtuple $(S, s_0, I, O, \delta, \lambda)$ where :*

- *S is a finite set of internal states,*
- *$s_0 \in S$ is the initial state,*
- *I is a set of actions, representing the input alphabet,*
- *O is the set of outputs,*
- *$\delta : S \times I \to S$ is a total state transition function,*
- *$\lambda : S \times I \to O$ is a total output function.*

An FSM starts in the initial state s_0 and accepts a word (a sequence of actions of its input alphabet) in order to produce an equally-sized sequence of outputs. State transition- δ and output function λ determine the next state and the output of an FSM upon receiving a single input. For each $s, s' \in S$, $i \in I$, and $o \in O$, we write $s \xrightarrow{i/o} s'$ when $\delta(s, i) = s'$ and $\lambda(s, i) = o$.

State transitions are extended inductively from a single input $i \in I$, to a sequence of inputs $w \in I^*$, i.e., we define $\delta(s, \epsilon) = s$ and $\lambda(s, \epsilon) = \epsilon$ where ϵ is the empty sequence; and for $s \in S, w \in I^*$,

and $a \in I$, we have $\delta(s, wa) = \delta(\delta(s, wa), a)$ and $\lambda(s, wa) = \lambda(s, w)\lambda(\delta(s, w), a)$, where juxtaposition of sequences denotes concatenation. For the sake of conciseness, we write $\delta(w)$ and $\lambda(w)$ instead of $\delta(s_0, w)$ and $\lambda(s_0, w)$.

In much of the literature in active learning, the system under learning is assumed to be complete and deterministic and we follow this common assumption in Definition 1 by requiring the state transition and output relations to be total functions. While the determinism assumption is essential for our forthcoming results to hold, we expect that the existing recipes for learning non-deterministic state machines can be made compositional using a similar approach as ours.

3.2 (De)Composing FSMs

Our aim is to produce a compositional learning algorithm for systems composed of interleaving parallel components, defined below. Due to the interleaving nature of parallel composition and determinism of the system under learning, the alphabets of these components are assumed to be disjoint.

Definition 2. *(Interleaving Parallel Composition) For two FSMs $M_i = (S_i, s_{0_i}, I_i, O_i, \delta_i, \lambda_i)$, with $i \in \{0, 1\}$, where $I_0 \cap I_1 = \emptyset$, the interleaving parallel composition of M_0 and M_1, denoted by $M_0 \mid\mid M_1$, is an FSM defined as*

$$(S_0 \times S_1, (s_{0_0}, s_{0_1}), I_0 \cup I_1, O_0 \cup O_1, \delta, \lambda)$$

where δ and λ are defined by

$$\delta((s_0, s_1), a) = \begin{cases} (\delta_0(s_0, a), s_1) \text{ if } a \in I_0, \\ (s_0, \delta_1(s_1, a)) \text{ otherwise, and} \end{cases} \qquad \lambda((s_0, s_1), a) = \begin{cases} \lambda_0(s_0, a) \text{ if } a \in I_0, \\ \lambda_1(s_1, a) \text{ otherwise.} \end{cases}$$

For $s_0 \in S_0$, $s_1 \in S_1$, and $a \in I_0 \cup I_1$

Next, we define the notions of projections for FSMs and for words; these notions are further used in the notion of (in)dependence and eventually in our proof of correctness to establish that the composed system has the same behaviour as the composition of the learned components.

Definition 3. *(Projection of an FSM) The projection of an FSM $M = (S, s_0, I, O, \delta, \lambda)$ on a set of inputs $I' \subseteq I$ denoted by $P(M, I')$, is an FSM $(S, s_0, I', O', \delta', \lambda')$, where*

- *$\delta'(s, a) = \delta(s, a)$ for $a \in I'$,*
- *$\lambda'(s, a) = \lambda(s, a)$ for $a \in I'$, and*
- *$O' = \{o \in O \mid \exists a \in I'.\ \exists s \in S.\ \lambda(s, a) = o\}$.*

Definition 4. *(Projection of a word) The projection of a word $w \in I^*$ on a set of inputs $I' \subseteq I$, denoted by $P_{I'}(w)$, is inductively defined as follows:*

$$P_{I'}(\epsilon) := \epsilon,$$
$$P_{I'}(au) := \begin{cases} aP_{I'}(u) \text{ if } a \in I', \\ P_{I'}(u) \text{ otherwise.} \end{cases}$$

Definition 5. *(Projection of an output sequence) The projection of the output sequence $w = o_1 \dots o_n$ with respect to an equally-sized sequence of inputs $v = i_1, \dots, i_n \in I^*$ and a subset of inputs $I' \subseteq I$, denoted by $P_{I'}(w, v)$, is defined as follows:*

$$P_{I'}(\epsilon, \epsilon) := \epsilon,$$
$$P_{I'}(ow, av) := \begin{cases} oP_{I'}(w, v) \text{ if } a \in I', \\ P_{I'}(w, v) \quad \text{otherwise.} \end{cases}$$

Definition 6. *((In)Dependent Actions) Consider an FSM M with a set of inputs I. The subsets $I_0, \ldots, I_n \subseteq I$ form an independent partition of I when for any $u \in I^*$, $\lambda_{P(M,I_0)||\ldots||P(M,I_n)}(u) = \lambda_M(u)$. Two inputs $i_0, i_1 \in I$ are independent when they belong to two distinct subsets of an independent partition. Two input actions are dependent, when they are not independent.*

Example. The partition $\big\{\{a\}, \{b\}, \{c, d\}\big\}$ in Figure 1(a) is not an independent partition because $\lambda_M(ab) = 10$ but $\lambda_{P(M,\{a\})||P(M,\{b\})||P(M,\{c,d\})}(ab) = 11$.

It immediately follows from Definition 6 and associativity of parallel composition (with respect to trace equivalence) that any coarser partitioning based on an independent partition is also an independent partitioning; this is formalised in the following corollary.

Corollary 1. *By combining two or more sets of an independent partition, the resulting partition remains independent.*

Moreover, it holds that any smaller subset of an independent partitioning is also an independent partitioning of the original state machine projected on the alphabet of the smaller subset, as specified and proven below.

Lemma 1. *Consider an independent partition $I_0, \ldots, I_n$ of inputs I for an FSM M; then for $K \subseteq \{0, \ldots, n\}$, $\{I_i \mid i \in K\}$ is an independent partition for $P(M, \bigcup_{i \in K}(I_i))$.*

Proof. Consider any subset $K \subseteq \{0, \ldots, n\}$ and $\{I_i \mid i \in K\}$ and consider any input sequence $u \in (\bigcup_{i \in K} I_i)^*$. Since u does not contain a symbol that is in any I_j for $j \notin K$, we have that $\lambda_{||_{i \in K} P(M,I_i)}(u) = \lambda_{P(M,I_0)||\ldots||P(M,I_n)}(u)$. Since $I_0, \ldots, I_n$ are independent, it follows likewise that $\lambda_{P(M,I_0)||\ldots||P(M,I_n)}(u) = \lambda_M(u)$. Using again that u has no symbol in any I_j for $j \notin K$, we know that $\lambda_M(u) = \lambda_{P(M,\bigcup_{i \in K}(I_i))}(u)$. Hence, $\lambda_{||_{i \in K} P(M,I_i)}(u) = \lambda_{P(M,\bigcup_{i \in K}(I_i))}(u)$, which was to be shown.

■

Lemma 2. *For any independent partition $I_0, \ldots, I_n \subseteq I$, $w \in I^*$ and $0 \leq i \leq n$, and state s it holds that $P_{I_i}(\lambda_M(s, w), w) = \lambda_{P(M,I_i)}(s, P_{I_i}(w))$.*

Proof. The proof uses induction on the length of w. Instead of proving the thesis, we prove the following stronger statement, which is possible because M can be viewed as the parallel construction of independent components.

$$P_{I_i}(\lambda_M((s_0, \ldots s_n), w), w) = \lambda_{P(M,I_i)}((s'_0, \ldots, s'_n), P_{I_i}(w)) \text{ with } s_i = s'_i.$$

Note that the lemma directly follows from this. Below we write $\vec{s}$ for $s_0, \ldots, s_n$, and likewise for $\vec{s}'$ and $\vec{s}''$.

The base case ($|w| = 0$) holds trivially as $w = \epsilon$. For the induction step we assume that the induction hypothesis holds for $|w| = k$ and we show that it holds for $w' = aw$ for arbitrary $a \in I$.

We first consider the case where $a \notin I_i$. We derive

$$\begin{aligned} P_{I_i}(\lambda_M(\vec{s}, aw), aw) &= P_{I_i}(\lambda_M(\vec{s}, a)\lambda_M(\delta(\vec{s}, a), w), aw) && \text{Definition 1} \\ &= P_{I_i}(\lambda_M(\delta(\vec{s}, a), w), w) && \text{Definition 5.} \\ &= \lambda_{P(M,I_i)}(\vec{s}', P_{I_i}(w)) && \text{Induction hypothesis.} \\ &= \lambda_{P(M,I_i)}(\vec{s}'', P_{I_i}(aw)) && \text{Definition 4.} \end{aligned}$$

By construction the i-th state in $\delta(\vec{s}, a)$ is equal to s_i as $a \notin I_i$. Hence, using the induction hypothesis, $\vec{s}'_i = s_i$. By definition $\vec{s}' = \delta(\vec{s}'', a)$ and hence, $\vec{s}''_i = \vec{s}'_i = \vec{s}_i$ as we had to show.

The other case we must consider is $a \in I_i$. Again the derivation is straightforward.

$$\begin{aligned} P_{I_i}(\lambda_M(\vec{s}, aw), aw) &= P_{I_i}(\lambda_M(\vec{s}, a)\lambda_M(\delta(\vec{s}, a), w), aw) && \text{Definition 1} \\ &= \lambda_M(\vec{s}, a)P_{I_i}(\lambda_M(\delta(\vec{s}, a), w), w) && \text{Definition 5.} \\ &= \lambda_M(\vec{s}', a)\lambda_{P(M,I_i)}(\delta(\vec{s}', a), P_{I_i}(w)) && \text{Induction hypothesis.} \\ &= \lambda_{P(M,I_i)}(\vec{s}, P_{I_i}(aw)) && \text{Definition 4.} \end{aligned}$$

Using the induction hypothesis it follows that $s_i = s'_i$, which concludes the proof. ■

3.3 Model Learning

Active model learning, introduced by Dana Angluin, was originally designed to formulate a hypothesis $\mathscr{H}$ about the behavior of a System Under Learning (SUL) as an FSM. Model learning is often described in terms of the Minimally Adequate Teacher (MAT). In the MAT framework, there are two phases: (i) hypothesis construction, where a learning algorithm poses Membership Queries (MQ) to gain knowledge about the SUL using reset operations and input sequences; and (ii) hypothesis validation, where based on the model learned so far, the learner proposes a hypothesis $\mathscr{H}$ about the "language" of the SUL and asks Equivalence Queries (EQ) to test it. The results of the queries are organised in an observation table. The table is iteratively refined and is used to formulate $\mathscr{H}$.

Definition 7. *(Observation Table) An observation table is a triple (S, E, T), where $S \subseteq I^*$ is a prefix-closed set of input strings (i.e., prefixes); $E \subseteq I^+$ is a suffix-closed set of input strings (i.e., suffixes); and T is a table where rows are labeled by elements from $S \cup (S.I)$, columns are labeled by elements from E, such that for all $pre \in S \cup (S.I)$ and $suf \in E$, $T(pre, suf)$ is the SUL's output suffix of size $|suf|$ for the input sequence $pre.suf$.*

The L* algorithm initially starts with S only containing the empty word ϵ, and E equals set of inputs alphabet I. Two crucial properties of the observation table, closedness and consistency, defined below, allow for the construction of a hypothesis.

Definition 8. *(Closedness Property) An observation table is closed iff for all $w \in S.I$ there is a $w' \in S$ that for all $suf \in E$, $T(w, suf) = T(w', suf)$ holds.*

Definition 9. *(Consistency Property) An observation table is consistent iff for all $pre_1, pre_2 \in S$, if for all $suf \in E$, $T(pre_1, suf) = T(pre_2, suf)$, it holds that $T(pre_1.\alpha, suf) = T(pre_2.\alpha, suf)$ for all $\alpha \in I, suf \in E$.*

MQs are posed until these two properties hold, and once they do, a hypothesis $\mathscr{H}$ is formulated. After formulating $\mathscr{H}$, L* works under the assumption that an EQ can return either a counter-example (CE) exposing the non-conformance, or yes, if $\mathscr{H}$ is indeed equivalent to the SUL. When a `CE` is found, a `CE` processing method adds prefixes and/or suffixes to the observation table and hence refines $\mathscr{H}$. The aforementioned steps are repeated until EQ confirms that $\mathscr{H}$ and SUL are the same. In between MQs, we often need to bring the FSM back to a known state; this is done through reset operations, which are one of our metrics for measuring the efficiency of the algorithm. EQs are posed by running a large number of test-cases and hence they are (two- to three) orders of magnitude larger than MQs. These test cases are generated through a random-walk of the graph or through a deterministic algorithm that tests all states and transitions for a given fault model. Two examples of deterministic test-case generation algorithms are the W- and WP-method [7]. It appears from recent empirical evaluations that for realistic systems deterministic equivalence queries are not efficient [4].

Since we are going to be learning the system in terms of components with disjoint alphabets, we define the following projection operator that removes all the transitions that are not in the projected alphabet. Our compositional learning algorithm basically learns a black-box with respect to its projection on the actions available in each purported component.

Definition 10. *(L^* with projected alphabet) Given an SUL $M = (S, s_0, I, O, \delta, \lambda)$ and $I' \subset I$, $L^*(M, I')$ returns $P(M, I')$ by running algorithm L^* with projected alphabet I' on M.*

4 Compositional Active Learning

In this section, we present an algorithm that learns the SUL in separate components and uses the interleaving parallel composition of the learned components to reach the total behavior of the system. Each component has an input alphabet I_i, which is disjoint from the alphabet of all the other components. The set of the input alphabets of components $I^F = \{I_1, \ldots, I_n\}$ is a partition of the total system's input alphabet. The main idea is to find an independent partitioning I^F. To reach such a partitioning, we start with a partition with singleton sets and iteratively merge those sets that are found to be dependent on each other. Then for $I_i \in I^F$, we learn the SUL with the projected alphabet I_i, and compute the product of the obtained components with interleaving parallel composition. The result is equivalent to the SUL if I^F is an independent partition.

Definition 11. *(LearnInParts) The LearnInParts function gets $M = (S, s_0, I, O, \delta, \lambda)$ and the partition $I^F = \{I_1, \ldots, I_n\}$ of I and returns the interleaving parallel composition of the learned components.*

$$LearnInParts(M, I^F) = L^*(M, I_1) \ || \ldots || \ L^*(M, I_n).$$

Definition 12. *(Composition) Given a partition $I^F = \{I_1, \ldots, I_n\}$ and $D \subseteq \{1, \ldots, n\}$, the Composition of I^F over D merges all the I_i ($i \in D$) in I^F.*

$$Composition(I^F, D) = (I^F \setminus \{I_i | i \in D\}) \cup \{ \bigcup_{i \in D} I_i \}.$$

Example. If $I^F = \{\{a\}, \{b\}, \{c\}, \{d\}\}$ and $D = \{1, 3, 4\}$, then $Composition(I^F, D) = \{\{a, c, d\}, \{b\}\}$.

Algorithm 1: Compositional Learning Algorithm (CL*)

```
   Result: 𝒦
1  Input: I^F = {I_1, ..., I_n}, M
2  𝒦 ← LearnInParts(M, I^F)
3  eq ← Equivalence-Query(𝒦, M)
4  while eq ≠ yes do
5  |  CE ← eq
6  |  D ← InvolvedSets(CE, I^F)
7  |  I^F ← Composition(I^F, D)
8  |  𝒦 ← LearnInParts(M, I^F)
9  |  eq ← Equivalence-Query(𝒦, M)
10 end
11 return 𝒦, I^F
```

Definition 13. *(InvolvedSets) The function InvolvedSets gets a counter-example* CE *and a partition* $I^F = \{I_1, \ldots, I_n\}$ *and returns indices of the sets in* I^F *that contains at least one character of* CE*:*

$$InvolvedSets(\texttt{CE}, I^F) = \{j \mid I_j \in I^F,\ \exists i\ \texttt{CE}[i] \in I_j\},$$

where the i^{th} *character of* CE *is denoted by by* $\texttt{CE}[i]$*.*

The function *InvolvedSets* allows us to detect some dependent sets by using a minimal counter-example since all actions in the counter-example are dependent, as we prove in Theorem 2.

Algorithm 1 shows the pseudo-code of the compositional learning algorithm. Initially the algorithm is called with the singleton partitioning I^F of the alphabet I and the SUL M, i.e., if the input alphabet is $I = \{a_1, a_2, \ldots, a_n\}$, then the initial partition of the alphabet will be $I^F = \{\{a_1\}, \{a_2\}, \ldots, \{a_n\}\}$. The *LearnInParts* method on line 2 learns each of the components given the corresponding alphabet set using the algorithm L* and returns the interleaving parallel composition of the learned components. If the oracle (MAT) returns yes for the equivalence query regarding hypothesis $\mathscr{H}$, the algorithm terminates and returns $\mathscr{H}$. Otherwise an(other) iteration of the loop is performed. The *InvolvedSets* method in line 6 extracts the dependent sets from the counter-example returned by the oracle; subsequently, *Composition* merges those sets into one. The *LearnInParts* method in line 8 is run again and the loop continues until the correct hypothesis is learned. We assume that the oracle always returns a minimal counter-example; this assumption is used in the proof of soundness (Theorem 2).

4.1 Termination Analysis

To prove the termination of our algorithm, we start with the following lemma which indicates how the counter-example is used to merge the partitions.

Lemma 3. *Let* $I^F = \{I_1, \ldots, I_m\}$ *be a partition of the system's input alphabet. If the teacher responds with a counter-example* CE*, then there are at least two actions* $u \in I_i, v \in I_j$ *in* CE *such that* $I_i \neq I_j \ \wedge\ I_i, I_j \in I^F$*.*

Proof. We prove this by contradiction. Suppose CE consists of actions that all belong to I_i. Let $C_i = L^*(M, I_i)$ with output function λ_{C_i}. Since the output of L* is always the correctly learned FSM

of the SUL, $\lambda_M(\texttt{CE}) = \lambda_{C_i}(\texttt{CE})$. Also, since C_i is a component of $\mathscr{H}$ produced by *LearnInParts*, $\lambda_{\mathscr{H}}(\texttt{CE}) = \lambda_{C_i}(\texttt{CE})$ based on Definition 2. This means CE can not be a counter-example. ■

The next lemma uses Lemma 3 to show how counter-examples will ensure progress in the algorithm, eventually guaranteeing termination.

Lemma 4. *At each round of the algorithm CL*, $|I^F|$ decreases by at least 1.*

Proof. By Lemma 3, at each round of the algorithm, at least two dependent sets are found by InvolvedSets, and the algorithm merges these dependent sets into a single set. Thus the size of the partition decrements by at least one; hence, the lemma follows. ■

Now we have the necessary ingredients to prove termination below.

Theorem 1. *The Compositional Learning Algorithm terminates.*

Proof. Assume, towards contradiction, that the algorithm does not terminate. Let I be the alphabet, an I_k^F be the partition of I after the k^{th} round of the algorithm. By Lemma 4, after at least $k = |I| - 1$ rounds, $|I_k^F| = 1$. Also by the assumption, the algorithm has not terminated at round k. Since $I_k^F = I$, the algorithm reduces to algorithm L* which terminates. Hence, the contradiction. ■

We prove next that every time we merge two partitions, there is a sound reason (i.e., dependency of actions) for it.

Theorem 2. *Let* CE *be the minimal counter-example returned by the oracle at round k of the algorithm and $I^F = \{I_1, \ldots, I_n\}$ the partition of the alphabet at the same round. Then, all actions in* CE *are dependent.*

Proof. Let $\texttt{CE} = wa$, $w \in I^*$ and $a \in I$, and $d = \{d_1, \ldots, d_m\}$ be an independent partition for the SUL M. Assume some actions in w are independent from a (proof by contradiction). Let d_k be the set in d that includes a. The set $I \setminus d_k$ contains all the independent actions from a. For M, we define $O_M = P_{d_k}(\lambda_M(wa))$; according to Lemma 2, $O_M = \lambda_{P(M,d_k)}(P_{d_k}(wa))$. The algorithm makes the hypothesis $\mathscr{H} = P(M, I_1)||\ldots||P(M, I_n)$ at the current round k. Since d_k is the union of a subset of I^F (algorithm has not terminated yet), $O_{\mathscr{H}} = P_{d_k}(\lambda_{\mathscr{H}}(wa)) = \lambda_{P(\mathscr{H},d_k)}(P_{d_k}(wa))$. If $O_{\mathscr{H}} \neq O_M$, then $P_{d_k}(wa)$ is a smaller counter-example than wa, which is a contradiction. Otherwise if $O_{\mathscr{H}} = O_M$, given that wa is a counter-example, $P_{I\setminus d_k}(\lambda_M(wa)) \neq P_{I\setminus d_k}(\lambda_{\mathscr{H}}(wa))$; if so, $P_{I\setminus d_k}(wa)$ is a smaller counter-example, hence the contradiction. ■

By Theorems 2 and 1, we have shown that the algorithm detects the independent action sets and eventually terminates. The next theorem is formulated to show that it terminates as soon as all dependent action sets have been detected.

Theorem 3. *Let $I^F = \{I_1, \ldots, I_n\}$ be an independent partition of the alphabet at round k. The algorithm terminates in this round.*

Proof. We prove this by contradiction. Assume that the algorithm does not terminate, and CE is the minimal counter-example returned by the oracle. By theorem 2, *InvolvedSets* returns two or more dependent sets from I^F. Since all the elements in I^F are pairwise independent, we confront the contradiction. ■

4.2 Processing Counter-examples

As mentioned in Theorem 2, we require all the actions in a minimal counter-example returned by the oracle to be dependent. However, most equivalence checking methods do not find the minimal

Algorithm 2: CE distillation

```
   Result: CE_M
1  Input: I^F = {I_1, ..., I_n}, CE, M, ℋ
2  CE ← CutCE(CE)
3  D ← InvolvedSets(CE, I^F)
4  for k ∈ {2, ..., size(D)} do
5      C ← all k combinations(D)
6      while C is not empty do
7          I ← C.pop
8          A ← ⋃_{i∈I} I_i
9          CE_A ← P_A(CE)
10         if CE_A is a counter-example then
11             Return CE_A
12         end
13     end
14 end
```

counter-example. For a non-minimal counter-example, we define a process called "distillation", which asks a number of extra queries to find the dependent actions. It iteratively gets a subset of *InvolvedSets*(CE, I^F) in the order of their sizes and merges its members together, producing a set M. The algorithm introduces P_M(CE) as output if it is a counter-example.

Suppose CE is the counter-example returned by the oracle at round k of the algorithm, and I^F is the alphabet partition at that round. To distill two or more dependent sets from CE, we follow Algorithm 2. The function *CutCE* on line 2 takes a counter-example CE and returns the smallest prefix of CE, which is also a counter-example (i.e., the SUL and the hypothesis model produce different outputs for it). Then, iteratively, it gets a subset of *InvolvedSets*(CE, I^F) in the order of their sizes and merges its members together, producing set M. The algorithm returns P_M(CE) as output if it is a counter-example.

The cost of CE-distillation algorithms is exponential in terms of the size of CE in the worst case. However, in the results section, we show that in practice, the cost of this part is not very significant compared to the total cost of learning.

Theorem 4. *All actions in the output of the CE distillation algorithm are dependent.*

The proof is omitted as it is similar to the proof of Theorem 2.

5 Empirical Evaluation

In this section, we present the design and the results of the experiments carried out to evaluate our approach, in order to answer the following research questions:

RQ1 Does CL* require fewer resets, compared to L*?
RQ2 Does CL* require fewer input symbols, compared to L*?

As stated in Section 1, these two research questions measure the efficiency of a learning method in a machine-independent manner: the number of input symbols summarises the total cost of a learning

campaign, while the number of resets summarises one of its most costly parts. Note that although active learning processes are structured in terms of queries, the queries used in the processes have vastly different lengths and it has been observed earlier that the total number of input symbols is a more accurate metric for comparison of learning algorithms than the number queries [36].

5.1 Subject Systems

A meaningful benchmark for our method should feature systems of various state sizes and various numbers of parallel components and with a non-trivial structure that may require multiple learning rounds. Also, we would like to have realistic systems, so that our comparisons have meaningful practical implications.

To this end, we choose the Body Comfort System (BCS) [25], which is an automotive software product line (SPL) of a Volkswagen Golf model. This SPL has 27 components, each representing a feature that provides specific functionality. The transition system of each component is provided in a detailed technical report [24]. We use the finite state machines of the components constructed from the transition system representations in [35] and compose several random samples utilising the interleaving parallel composition (Definition 2) to build the product FSMs. We automatically constructed 100 FSMs consisting of a minimum of two and a maximum of nine components in this case study. The maximum number is chosen due the performance limits of L*; beyond this limit, our learning campaign for L* could take more than four hours. All experiments were conducted on a computer with an Intel® Core™ M-5Y10c CPU and and 8GB of physical memory running Ubuntu version 20 and LearnLib version 0.16.0. Our subject systems have a minimum of 300 states and a maximum of 3840 states, and their average number of states is 1278.2 with a standard deviation of 847. We started the calculation of the metrics for subject systems of at least 300 states, since for small subject systems, the advantage of compositional learning is not significant.

5.2 Experiment Design

To answer the research questions, we implemented the compositional learning algorithm on top of the LearnLib framework [30]. This implementation uses the equivalence oracle in two places; to learn projections in the *LearnInParts* function and to check the hypothesis/SUL equivalence. The performance of the algorithm significantly relies on the type of equivalence queries used by the underlying L* algorithm. We experimented with a number of equivalence methods and settled upon using random walks; when using deterministic algorithms such as the WP- and the WP-method, for large systems, the cost of equivalence queries becomes prohibitively high and obscures any gain obtained from compositionality. To ensure that our results are sound, we have carried out similar experiments by using an additional deterministic equivalence query at the end of the learning campaign, when the last random equivalence query does not return any counter-example. This additional step verifies our comparisons when an assurance about the accuracy of the learning process is required. More details about these additional experiments can be found in our public lab package [23] (`https://github.com/faezeh-lbf/CL-Star`).

We enabled caching, since caching significantly reduces repetitive queries. We repeat each learning process three times, comparing the number of resets and input symbols for L* and CL*.

In addition to reporting the median metrics, their standard deviations, and the relative percentage of improvements, we use the statistical T-test to answer the research questions with statistical confidence and report the p-values. We analyse the distribution of the results and establish their

normality using K-tests. We use the SciPy [20] library of Python to perform statistical analysis and Seaborn [38] for visualising the results.

5.3 Results

In this section, we first present the results of our experiments and use them to answer our research questions. Then we show how the number of components in an FSM affects the efficiency of our algorithm. Finally, we discuss threats to the validity of our empirical results.

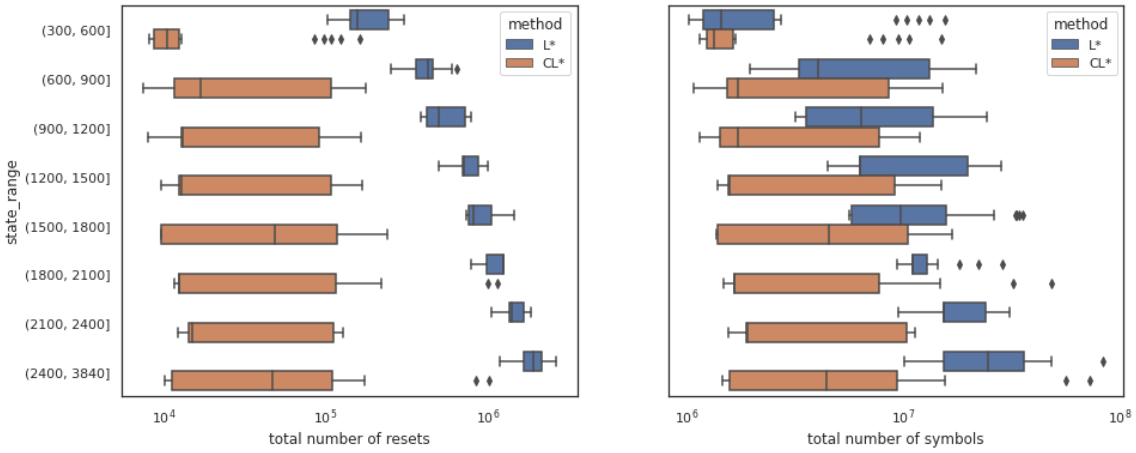

Fig. 2: The total number of input symbols and resets in the CL* and L* methods

We cluster the benchmark into eight categories based on the FSM's number of states and illustrate the distribution of input symbols and resets for each cluster in Figure 2. In this figure, the CL* and L* methods are compared based on the metrics mentioned. The scale of the x-axis (the value of metrics) is logarithmic.

Tables 1 and 2 summarise the results of our experiments. For each category, we calculate the median and standard deviation of our metrics (the number of input symbols and resets) both for L*

Table 1: Comparing the total number of input symbols in the CL* and L* methods

#States	L* method		CL* method		Progress	p-value
	Median	Standard deviation	Median	Standard deviation	percentage	(one-sided paired T-test)
(300, 600]	1443710	2834380.581	1329818	2382620.467	14.47	7.43e-3
(600, 900]	4013396	6262292.443	1716878.5	4408369.926	36.44	1.54e-8
(900, 1200]	6387472	6663334.645	1714934.5	3757307.024	52.37	8.36e-7
(1200, 1500]	6259466	9311767.302	1576494	4798094.639	57.28	6.49e-4
(1500, 1800]	9700935	10726103.24	4498072	5576873.639	54.58	4.30e-4
(1800, 2100]	11070428	5310108.013	1649557	13958718.62	37.51	2.96e-2
(2100, 2400]	15348181	6287714.182	1888226	4215184.514	70.80	1.80e-10
(2400, 3840]	24700222.5	14837416.08	4385086	13817389.06	68.42	2.66e-12

Table 2: Comparing the total number of resets in the CL* and L* methods

#States	L* method		CL* method		Progress	p-value
	Median	Standard deviation	Median	Standard deviation	percentage	(one-sided paired T-test)
(300, 600]	157971	65257.85738	10433	28259.60196	90.46	1.05e-33
(600, 900]	425260.5	77944.01883	16808	56274.51558	86.33	1.07e-43
(900, 1200]	501347.5	147915.8363	13109	50224.87222	90.87	3.80e-16
(1200, 1500]	712999	136904.04	12811	60125.8884	91.77	4.18e-13
(1500, 1800]	823482	275862.8299	48344	80507.59837	91.73	4.97e-13
(1800, 2100]	1262025	188390.1181	12412	369932.964	84.07	2.18e-06
(2100, 2400]	1412237	220211.8459	15042	53006.08784	95.83	2.44e-14
(2400, 3840]	1900234	427883.9888	46624.5	201052.8807	94.67	2.20e-23

and CL*. The metric "progress percentage" is defined to measure the improvement brought about by compositional learning (compared to L*). For each metric, the progress percentage is calculated as $(1 - \frac{p}{q}) * 100$, where p and q are the value of that metric in CL* and L*, respectively. A positive progress percentage in a metric shows that the CL* is more efficient in terms of that metric. To measure the statistical significance, we used the one-sided paired sample T-test to check if there was a significant difference ($p < 0.05$) between the metrics in the two algorithms.

Both Tables 1 and 2 indicate major improvements, particularly for large systems, in terms of the total number of input symbols and resets, respectively. Compositional learning reduces the number of symbols up to 70.80 percent and the number of resets up to 95.83 percent. The statistical tests also confirm this observations and the p-values obtained from the tests are in all cases very low; in case of the number of input symbols the p-values range from 10^{-2} to 10^{-12}, while for resets they range from 10^{-6} to 10^{-43}, which are well-below the usual statistical p-values (0.05) and represent a very high statistical significance.

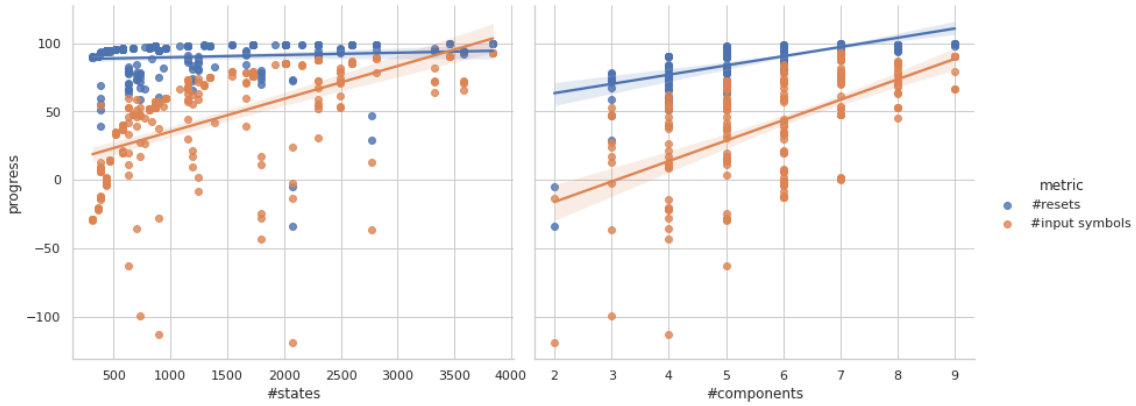

Fig. 3: The diagrams of improvement brought about by compositional learning vs. size of the SUL in terms of number states (left) and components (right).

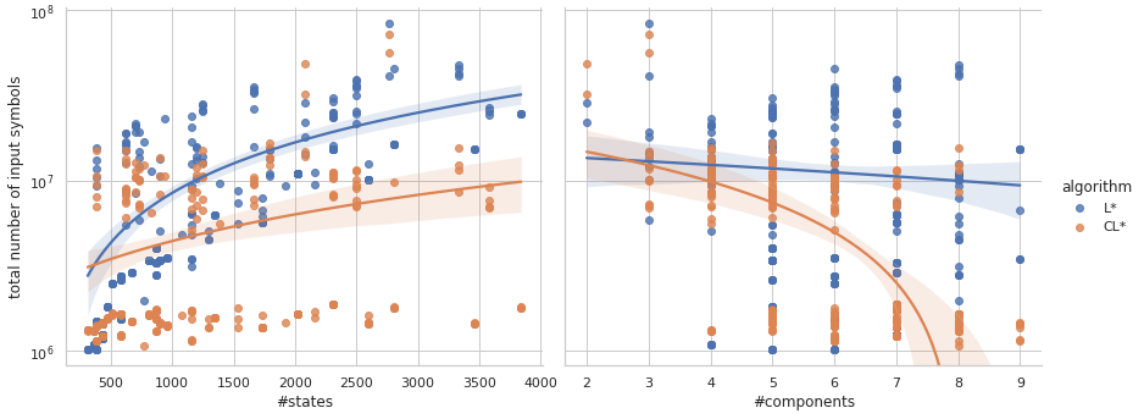

Fig. 4: The effect of FSM sizes in terms of the number of components and states on the total number of input symbols.

The plots in Figure 3 visualise the improvements brought about by compositional learning. This plot demonstrates that the saving due to compositional learning increases as the number of components in SULs increases. We further analysed the trends of our measured metrics in terms of the number of states and the number of parallel components. These trends are depicted for the total number of input symbols in Figure 4 and for the number of resets in Figure 5, respectively. These figures indicate that the increase of both metrics with the number of states is more moderate for the compositional learning approach, i.e., compositional learning is more scalable. More importantly, the right-hand-side of both figures signifies the effect of compositional learning when the number of parallel components increases while the number of states remains fixed.

Figure 6 shows the effect of the number of components on the total number of input symbols for a fixed state-space size for algorithms L* and CL*. In this plot, as the number of components increases, the corresponding dot will become darker and larger. According to this figure, the learning cost is lower for SULs with more components in both L* and CL*. Still, for CL* (the right side), the cost of learning SULs with more components is significantly lower because we structurally learn these components essentially independently.

As mentioned in Section 4.2, the cost of the CE distillation process can increase exponentially in the size of the counter-example. However, in practice, it seems to be much more tractable. To evaluate this, we count the number of input symbols required by the CE distillation process to learn each SUL. The median value of this metric is 1961 input symbols, which is insignificant compared the total cost of learning. In fact, the cost of CE distillation process for each group in Table 1 is between 0.037 and 0.12 percent of the total learning cost; the reported total learning cost (total number of input symbols) includes the cost of CE distillation.

5.4 Threats to Validity

In this section, we summarise the major threats to the validity of our empirical conclusions. First, we analyse the threats to conclusion validity, i.e., whether the empirical conclusions necessarily follow from the experiments carried out. Then, we discuss the threats to external validity concerning the generalisation of our results to other systems.

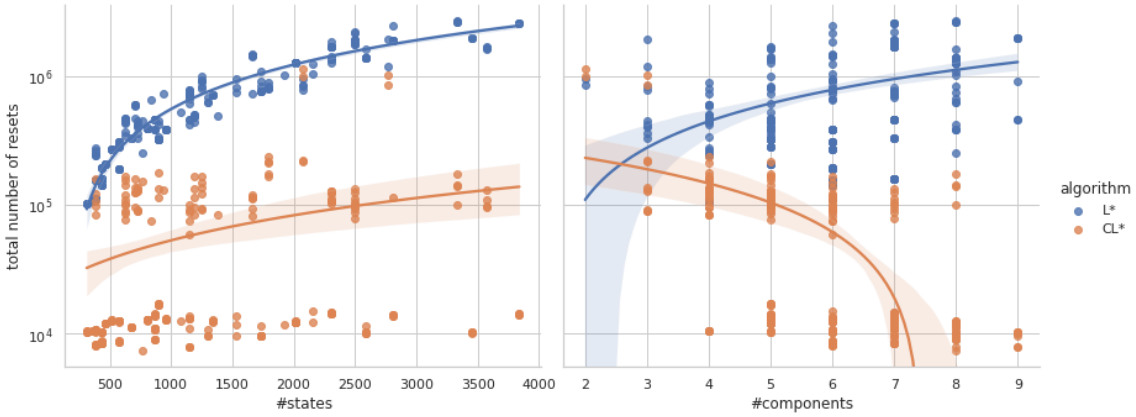

Fig. 5: The effect of the size of FSMs in terms of the number of components and states on the total number of required input resets.

We mitigated conclusion validity threats by using statistical tests to ensure that our observations (both in terms of improvement percentages in Tables 1 and 2 and the visual observations in Figures 2) do represent a statistically significant improvement. We opt for one-sided paired sample T-tests in order to minimise the threats to conclusion validity. We only conclude that the CL* is more efficient than the L* when there is a meaningful difference ($p < 0.05$) between the results of L* and CL*. To make sure that the chosen statistical test is applicable, we analysed the distribution of the data first.

We mitigated the risk of conclusion validity by using subject systems that are based on practical systems rather than using randomly generated FSMs. However, further research is needed to analyse the performance of our approach based on other benchmarks from other domains. We also mitigated the effect of using random equivalence queries by repeating the experiments with a final deterministic query.

6 Conclusions

In this paper, we presented a compositional learning method based on Angluin's algorithm L* that detects and independently learns interleaving parallel components of the system under learning. We proved that our algorithm, called CL*, is correct and we empirically showed that it causes significant gains in the number of input symbols and the number of resets in a learning campaign. The gain is significantly increased with the number of parallel components.

Our algorithm is naturally amenable to parallelisation and developing a parallel implementation is a natural next step. A more thorough investigation of counter-example processing in order to efficiently find a minimal counter-example is an area of further research, particularly, in the light of the recent results in this area [13]. Finding a trade-off between using deterministic and random (or mutation-based) equivalence queries is another area of future research. We would also like to investigate the possibility of developing equivalence queries that take the structure of the systems into account: we have observed that much of the effort in the final equivalence query (on the composed system) is redundant and the final equivalence query can be made much more efficient by

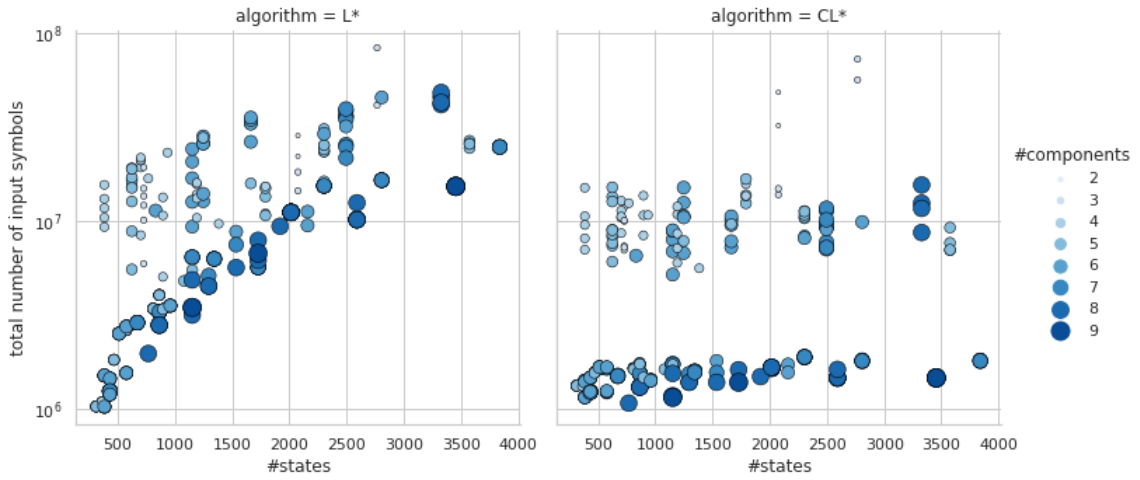

Fig. 6: The relation between the total number of symbols and the number of states and components for the algorithms L* (left) and CL* (right).

only considering the dependencies among purportedly independent partitions. Finally, extending our notion of parallel composition to allow for a possible synchronisation of components is another direction of future work; we believe inspirations from concurrency theory and in particular, Milner and Moller's prime decomposition theorem [26] may prove effective in this regard. Independently from our work, Neele and Sammartino [29] proposed an approach to learn synchronous parallel composition, under the assumption of knowing the alphabets of the components. This is a promising approach to incorporate synchronous parallel composition into our framework.

Acknowledgments

We would like to thank Rasta Tadayon and Amin Asadi Sarijalou for their contributions to the early stages of this work. The work of Mohammad Reza Mousavi was supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/2. We thank the reviewers of FOSSACS for their insightful and constructive comments, which, in our view, led to improvements in our final paper. We thank the Artifact Evaluation committee at ESOP/FOSSACS for their careful review of our lab package.

References

1. Aarts, F., de Ruiter, J., Poll, E.: Formal models of bank cards for free. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013. pp. 461–468. IEEE Computer Society (2013). https://doi.org/10.1109/ICSTW.2013.60
2. Aarts, F., Schmaltz, J., Vaandrager, F.W.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece,

October 18-21, 2010, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6415, pp. 673–686. Springer (2010). https://doi.org/10.1007/978-3-642-16558-0_54
3. Aichernig, B.K., Tappler, M.: Efficient active automata learning via mutation testing. Journal of Automated Reasoning **63**(4), 1103–1134 (2019). https://doi.org/10.1007/s10817-018-9486-0
4. Aichernig, B.K., Tappler, M., Wallner, F.: Benchmarking combinations of learning and testing algorithms for active automata learning. In: Ahrendt, W., Wehrheim, H. (eds.) Tests and Proofs - 14th International Conference, TAP@STAF 2020, Bergen, Norway, June 22-23, 2020, Proceedings [postponed]. Lecture Notes in Computer Science, vol. 12165, pp. 3–22. Springer (2020). https://doi.org/10.1007/978-3-030-50995-8_1
5. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 444–462. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_25
6. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
7. Broy, M., Jonsson, B., Katoen, J., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004], Lecture Notes in Computer Science, vol. 3472. Springer (2005). https://doi.org/10.1007/b137241
8. Cifuentes, C., Simon, D.: Procedure abstraction recovery from binary code. In: Proceedings of the Fourth European Conference on Software Maintenance and Reengineering. pp. 55–64. IEEE (2000). https://doi.org/10.1109/CSMR.2000.827306
9. Damasceno, C.D.N., Mousavi, M.R., da Silva Simão, A.: Learning by sampling: learning behavioral family models from software product lines. Empir. Softw. Eng. **26**(1), 4 (2021). https://doi.org/10.1007/s10664-020-09912-w
10. al Duhaiby, O., Groote, J.F.: Active learning of decomposable systems. In: Bae, K., Bianculli, D., Gnesi, S., Plat, N. (eds.) FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020. pp. 1–10. ACM (2020). https://doi.org/10.1145/3372020.3391560
11. Fiterau-Brostean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017. pp. 142–151. ACM (2017). https://doi.org/10.1145/3092282.3092289
12. Frohme, M., Steffen, B.: Compositional learning of mutually recursive procedural systems. Int. J. Softw. Tools Technol. Transf. **23**(4), 521–543 (2021). https://doi.org/10.1007/s10009-021-00634-y
13. Frohme, M., Steffen, B.: From languages to behaviors and back. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13560, pp. 180–200. Springer (2022). https://doi.org/10.1007/978-3-031-15629-8_11
14. Garhewal, B., Vaandrager, F.W., Howar, F., Schrijvers, T., Lenaerts, T., Smits, R.: Grey-box learning of register automata. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12546, pp. 22–40. Springer (2020). https://doi.org/10.1007/978-3-030-63461-2_2
15. Hooimeijer, B., Geilen, M., Groote, J.F., Hendriks, D., Schiffelers, R.R.H.: Constructive model inference: Model learning for component-based software architectures. In: Fill, H., van Sinderen, M., Maciaszek, L.A. (eds.) Proceedings of the 17th International Conference on Software Technologies, ICSOFT 2022, Lisbon, Portugal, July 11-13, 2022. pp. 146–158. SCITEPRESS (2022). https://doi.org/10.5220/0011145700003266
16. Howar, F., Steffen, B.: Active automata learning in practice - an annotated bibliography of the years 2011 to 2016. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany,

April 24-27, 2016, Revised Papers. Lecture Notes in Computer Science, vol. 11026, pp. 123–148. Springer (2018). https://doi.org/10.1007/978-3-319-96562-8_5
17. Howar, F., Steffen, B.: Active automata learning as black-box search and lazy partition refinement. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning : Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday, pp. 321–338. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-15629-8_17
18. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. Machine Learning **96**(1), 65–98 (2014). https://doi.org/10.1007/s10994-013-5419-7
19. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_26
20. Jones, E., Oliphant, T., Peterson, P.: Scipy: Open source scientific tools for python (01 2001). https://doi.org/10.1038/s41592-019-0686-2
21. Kearns, M.J., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (1994). https://doi.org/10.7551/mitpress/3897.001.0001
22. Koschke, R.: Architecture Reconstruction, p. 140–173. Springer-Verlag, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-540-95888-8_6
23. Labbaf, F., Groot, J.F., Hojjat, H., Mousavi, M.R.: Compositional Learning for Interleaving Parallel Automata (CL-Star) (Apr 2023). https://doi.org/10.5281/zenodo.7624699, https://doi.org/10.5281/zenodo.7624699
24. Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., Schaefer, I.: Delta-oriented test case prioritization for integration testing of software product lines. In: Proceedings of the 19th International Conference on Software Product Line. p. 81–90. SPLC '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2791060.2791073
25. Lity, S., Lachmann, R., Lochau, M., Schaefer, I.: Delta-oriented software product line test models-the body comfort system case study. Tech. Rep. 2012-07, TU Braunschweig (2012)
26. Milner, R., Moller, F.: Unique decomposition of processes. Theoretical Computer Science **107**(2), 357–363 (1993). https://doi.org/10.1016/0304-3975(93)90176-T, https://www.sciencedirect.com/science/article/pii/030439759390176T
27. Moerman, J.: Learning product automata. In: Unold, O., Dyrka, W., Wieczorek, W. (eds.) Proceedings of The 14th International Conference on Grammatical Inference 2018. Proceedings of Machine Learning Research, vol. 93, pp. 54–66. PMLR (feb 2019), https://proceedings.mlr.press/v93/moerman19a.html
28. Naeem Irfan, M., Oriat, C., Groz, R.: Model inference and testing. Advances in Computers, vol. 89, pp. 89–139. Elsevier (2013). https://doi.org/10.1016/B978-0-12-408094-2.00003-5, https://www.sciencedirect.com/science/article/pii/B9780124080942000035
29. Neele, T., Sammartino, M.: Compositional Automata Learning of Synchronous Systems. In: Lambers, L., Uchitel, S. (eds.) FASE 2023. Lecture Notes in Computer Science, Springer (2023)
30. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) Fundamental Approaches to Software Engineering. pp. 377–380. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1145/1081180.1081189
31. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021
32. Sanchez, L., Groote, J.F., Schiffelers, R.R.H.: Active learning of industrial software with data. In: Hojjat, H., Massink, M. (eds.) Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11761, pp. 95–110. Springer (2019). https://doi.org/10.1007/978-3-030-31517-7_7
33. Smeenk, W., Moerman, J., Vaandrager, F.W., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France,

November 3-5, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9407, pp. 67–83. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_5
34. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn – learning timed automata from tests. In: Formal Modeling and Analysis of Timed Systems: 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27–29, 2019, Proceedings. p. 216–235. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29662-9_13
35. Tavassoli, S., Damasceno, C.D.N., Khosravi, R., Mousavi, M.R.: Adaptive behavioral model learning for software product lines. In: Felfernig, A., Fuentes, L., Cleland-Huang, J., Assunção, W.K.G., Falkner, A.A., Azanza, M., Luaces, M.Á.R., Bhushan, M., Semini, L., Devroey, X., Werner, C.M.L., Seidl, C., Le, V., Horcas, J.M. (eds.) SPLC '22: 26th ACM International Systems and Software Product Line Conference, Graz, Austria, September 12 - 16, 2022, Volume A. pp. 142–153. ACM (2022). https://doi.org/10.1145/3546932.3546991
36. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (jan 2017). https://doi.org/10.1145/2967606
37. Vaandrager, F.W., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness. In: Fisman, D., Rosu, G. (eds.) Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2022. Lecture Notes in Computer Science, vol. 13243, pp. 223–243. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_12
38. Waskom, M.L.: seaborn: statistical data visualization. Journal of Open Source Software **6**(60), 3021 (2021). https://doi.org/10.21105/joss.03021